CRdata Programmer's Guide

Version 1.01 June 30, 2010

How to Read this Document

There are four colors in this document. Everything in black is explanatory text. Anything in pink is what the user actually types. Anything in green is pseudo code that does not actually get run but explains what the user should type in pink. Anything in brown is actual live code from CRdata.

Introduction

CRdata is an open source programming platform in which to develop R programs that can be run remotely on our resources or yours if you so choose to install your own version of CRdata. This document will guide you through the code and enable you to get your own system of CRdata up and running as smoothly as possible.

CRdata is written in the Ruby Programming language using the Ruby on Rails web development framework. We assume that you have some familiarity with the Ruby Programming Language and the web Framework Ruby on Rails. If you are not familiar with Ruby or Rails, you may want to take some time to get familiar as CRdata's code base is a direct reflection of the way all standard Rails applications are laid out.

CRdata Principles

REST

Representational State Transfer [REST] is at the heart of the way CRdata works. Every single request or event in the system uses REST for communication. Nothing can happen in CRdata without REST taking over and doing its work to make an event, call or signal communication move from an actor to a listener or from the client to the server. For that reason we start out the Programmer's Guide talking about REST and why it is so important to understand.

In order to understand Representational State Transfer [REST] it is best to go straight to the source to better understand what REST is all about. REST is interesting because the Internet is based on REST and the CRdata application is based on Ruby On Rails [ROR]. Ruby on Rails is an implementation of the REST protocol. The World Wide Web or what is commonly referred to as the Internet is an implementation of REST. So clearly REST is something important and for that reason it is very significant that programmer's trying to understand CRdata understand REST.

Roy Fielding's dissertation is one of the best known descriptions to clearly encapsulate the concept of software architecture. In fact, he devotes the whole first chapter of his paper to defining software architecture. The key to software architecture according to Fielding is the concept of abstraction. Any complex system will have many levels of abstraction with each level being encapsulated in order to sustain the boundaries. Its the recursive nature of these abstractions along with the associated interface that allows for systems to be decomposed into simpler and simpler components or elements.

Given the global world of all well defined software architectures one can begin to see software patterns emerge from a set of principles that are inherent in well defined model. The first important factor to understand when thinking about REST and more globally software architecture is the idea of constraining a system to limit the bounds of its possibilities. REST is based on a system of five main constraints which encapsulate its abilities and limit the architecture to well defined boundaries.

These constraints are in order --- Client-Server, Stateless, Cache, Interface, and Layers.

Fielding derives REST in his dissertation starting with what he calls the "NULL style" or what we would call "first principles" and so the ordering of the constraints are important for clarification of what is going on in REST.

Client-Server

Clients and Servers are separated by an Interface that is well defined. By decoupling the User Interface or Web Browser in the case of the web with the back end database storage system it enables future improvements and design on both sides of the equation without cluttering either side with the baggage of the other. In the case of the Internet clearly this has enabled competition because the vendors creating the client and the server are very different companies or organizations and they all only have to worry about speaking the same underlying architecture called REST.

Stateless

A constraint is now added to the client server which states that all requests from client to server must contain all of the information necessary for the server to understand the request from the client. By default this means that no information or STATE is stored on the server side about previous requests. The browser is able however to maintain session state and this is done in various way on different browsers but the main method for doing this is usually cookies.

One of the advantages to a Stateless system is that reliability of the system is improved because the server does not have to worry about recovering from partial failures. The reliability is purely a function of the database system tied to the back end server. Until the submit button is clicked and the transaction is completed in the database no money will be transferred from Bank Account A to Bank Account B and/or no credit card will be charged for buying a new book at Amazon.

The disadvantage of a Stateless system is that there is slightly more overhead in the system because each time a request is sent to the server the client needs to send all of the information not currently stored in the database each time and after a while this could possibly slow down the system if in fact there are thousands of people hitting the server at the same time. This is where the idea of layers and a proxy gets introduced into the architecture at a later point in time.

Cache

We now add a cache to the client-stateless-server style. The cache enables the web browser to store information on the client side that could possibly be used again in the future on another similar type of request. Whether or not data can be cached on the client side is a function of the server when it generates the response from the request putting in the response tags that tell the client whether or not it has permission to cache a particular block of data.

The advantage of caching a response is that less data in similar future requests have to be returned because the needed information is already there in your browser. The concept of caching in the REST architecture is almost identical to the concept of caching when looking at microprocessor design and the data bus.

Interface

Hypertext Transfer Protocol [HTTP], the interface of REST, is probably the most important feature of the REST architecture and what enabled the Internet to explode on the scene in the mid 1990's. Up until this time, user interfaces had custom protocols to talk to their back end servers. It was the first time in the history of modern computation that one stateless interface was declared and everyone adopted it. Furthermore, a set of standard interfaces is the main design pattern in any well architected software system.

Layer

Layering adds the ability to add intermediaries such as proxies between the client and the server. The importance of the layering system is that the client and/or the server does not know, and does not NEED to know whether there are proxies or other types of layers between the actual request from the browser and the response coming back from the server. Thats why when you make a request from your browser you don't need to worry at all about the route that your request takes and/or the route that comes back from the response. The round trip of the message can take many circuitous routes and all of them are legal in terms of REST because of the Layer constraint.

Design Patterns

Throughout the course of building things people start to realize that certain things can be used over and over again because they are functional units or elements of the thing being built. With cars people think of modules like engines and transmissions, with houses people think of windows, doors, and stairways and in software engineers think of design patterns as a solution that can be used in many different projects or applications.

The history of design patterns in Software Engineering dates back to the late 1970's but really took off after "The Gang of Four" (Gamma et al.) published their seminal book in 1994.

The list of design patterns is rather lengthy and becoming an expert on all the patterns takes most software engineers some time, but like most things in life, its the use of the patterns in real world software problems that solidifies the idea in your head.

In CRdata we use two main design patterns that we will explain next namely Model View Controller and Active Record. Roy Fielding's REST architecture can be considered an Architectural Pattern which is similar.

Model, View, Controller Design Pattern

CRdata is organized around the Model, View, Controller design pattern, here after called **MVC.** Most modern web applications today are based on MVC, and by choosing Ruby on Rails we get the MVC design pattern for free, its the way Rails works. The MVC pattern defines a programming structure which is well defined and specifies exactly where different parts of your code should reside.

The Model View Controller [MVC] design pattern was invented so that the graphical user interface of an application could be abstracted away from the underlying code that actually does the work. This solves the problem of decoupling the data or model in your system and the presentation or view. In CRdata, the model is the data stored in a database and the the view is either the HTML representation that you see in your browser or the XML representation that gets returned to an application like a Processing Node during a REST call.

One major advantage of the MVC pattern is the ability to have multiple views and controllers of the same data. In CRdata we currently have two views, the standard view you see in your browser and the REST view that gets returned to the Processing Node. We could very easily add a third view for your mobile phone and a fourth view for an application that displays data inside an applet or Windows application. Because of the separation of concern, new views and controllers can be added that interface with the same model without forcing a change in the model design.

The MVC pattern starts out or gets initiated when an Event or Request in the case of a Browser comes into the Controller which in turn will change a model, view, or both. If a controller changes a model, then all of the views associated with that model are automatically updated. Similarly, whenever a controller changes a view by revealing a hidden area with a check box, the view gets data from the underlying model to refresh itself. The model works by registering views and later notifying all of its registered views when any of its functions cause its state to be changed.

Active Record Design Pattern

Like Model View Controller [MVC], Active Record [AR] is another major design pattern deployed in CRdata. It is the mechanism by which data is stored in a relational database. It was named by Martin Fowler, a preeminent computer scientist, who has spent many years working with systems that store data. The beauty of AR is that it abstracts away the actual task of storing data in a relational database by defining a very simple interface that stores data. The interface sometimes known as CRUD, Create, Read, Update, Delete enables objects to be stored in a database with minimal work.

The underlying software that does the actual work of storing data into a database is rather complicated. Lots of details that the programmer need not be concerned with including magically creating long, complex, SQL strings that put data into the database, manipulate it while it is there, and delete it later. With one simple interface AR is the main mechanism for persisting information on the internet today. Other common terms used to describe this technology include Object Relational Mapping [ORM] and Data Access Object [DAO].

Each database table in CRdata is tied to a model which is encapsulated in a Ruby class that basically mimics the data representation and functionality or logic. Each instance of the model or class is called an object. In object oriented programming the class is the abstract representation and each object is the real live thing that lives in the run time environment. Thus an instance of the object is tied to a single row in a particular database table. So, if you want to create a new row in the table, you first instantiate the object and then call save on that object instance, once the database transaction completes, the data is permanently saved in the database forever, until at a later point in time when the data gets updated or deleted.

Active Record is the default model component in the MVC design pattern described above. It can also be used as a stand alone package for other Ruby applications not tied to Ruby on Rails [ROR].

CRdata Architecture

Model

The models correspond to the database tables in the CRdata system. A model is the data in the system and all of the algorithms associated and surrounding manipulating that data. All models in CRdata extend the class ActiveRecord::Base

It is important for you to understand **ActiveRecord** as this is how all of our data is persisted to the PostgreSQL database. ActiveRecord is the framework that provides **CRUD** create, read, update, and delete from the database. Each one of these tables or models is the core of the system. Everything emanates from here, and so it is important that fundamentally you understand what each model does.

Job

The goal of the model Job is to move the job through the different states as the job gets processed. Initially the job is in the Submitted state. The job will stay in the submitted state until there is a processing node ready to start working on the job. If you submit a job and it hangs around in the Submitted state for awhile that means that all of the processing nodes are busy doing other work. It will also stay in the Submitted state if for some reason there are no processing nodes running at the time your job gets submitted.

A job gets moved into the submitted state by the jobs_controller. The jobs_controller will move a job into the submitted state in either one of two cases. The first default case is when you submit a job to the queue from the UI. The second way a job will get moved into the submitted state is if you clone a job.

Data Set

The data_set is a particular data file that gets uploaded to the system. The data files are stored on S3, while R scripts are actually stored in the database.

Jobs Queue

This is the place where the jobs live once they get submitted. As jobs are submitted to CRdata we put them into a queue which is then accessed when a processing node becomes free.

Notifier

The notifier is the part of the system that notifies a user when their job has completed.

Processing Node

This is the computer process and or virtual machine that the job runs on. Each processing node must be registered by the user before it can be used. See the Client side Processing Node document for further details.

R Script

The actual underlying R script that the user submits or uses on the system. The R script is stored in the database where as the underlying data files get stored on S3.

View

The view is the user interface of the system or what is seen inside the browser. The nice part about the view is that depending on where you are rendering the UI, different HTML or XML code can be generated. In fact, if you look inside the CRdata code you will see methods that generate out XML or HMTL depending on whether the data is being returned as a REST call or HTML back to the browser. You could also imagine a third case where you are viewing CRdata job status on a cell phone or PDA and in that case the view would return something different. The key point here is that the User Interface code is completely decoupled from the data or model and the controller or glue code that ties the model and view together.

The view system in CRdata is based on a Rails package called **Action**

View.   Action View is in charge of rendering templates along with including nested and partial templates.

Each view in the system has its own directory under which a small set of files live which render the corresponding view. Action View templates in CRdata have two different types of file extension formats. The most common extension of a view file is ".erb" and a small number of files end in ".rjs". If the template file has a .erb extension then it uses a mixture of ERb (included in Ruby) and HTML. If the template file has a .rjs extension then it will used by Javascript.

Helper

Action View Helpers are optional objects that assist in creating the HTML that are created by Ruby on Rails but are not always used. There are two kinds of helper functions. The first being application-level helper functions which go in the top level ApplicationHelper which are accessible from any controller or view template. The other are helper functions that are specific to a particular controller go in their respective objects.

The following helpers are being used in our system.

- ApplicationHelper
- DataSetsHelper
- GroupsHelper
- JobsHelper
- JobsQueuesHelper
- ProcessingNodesHelper
- RScriptsHelper

Controller

The main job of a Controller is to tie a model and view together. The incoming request from the browser is handled by the controller. It then gets the data from the database or model, and passes that data on to the view for rendering in HTML which is then sent back to the browser as a response.

application_controller

The class ApplicationController extends ActionController::Base. All of the controller's in CRdata extend ApplicationController, a custom class which we wrote enabling us a layer of indirection between our controller classes and ActionController::Base.

The job of ActionController is to process incoming requests from the browser. It does this by extracting parameters from the request and then handing them off to the intended action. ActionController also does redirect management, template rendering and manages session state.

jobs_controller

One of the things the jobs_controller does is submit new jobs and clone existing jobs. It is also responsible for moving the job through its different state ultimately to completion or failure. Another thing the jobs_controller does is display all of the jobs in the system and the status of the job. If you are only interested in one particular job, the jobs_controller will show you the state of a job based on the job number.

The processing nodes talk to the job_controller through REST and tell it what state to put the job into. For example, a rest call from the processing node tells the jobs_controller with a boolean true or false whether the job completed successfully or not. Some of these REST calls are outlined in the file routes.db which we will talk more about later in another section of this document.

REST

CRdata at its essense is a distributed computing platform with processing nodes running on different computers talking back to an application server or web server that hands off work to them. All of the communication between the processing node and the server is enabled by REST and so it is important that you understand the fundamentals of this very interesting topic.

I am going to outline REST using Stefan Tilkov's Rest Principles which are:

- Give every “thing” an ID
- Use standard methods
- Communicate statelessly

Give every "thing" an ID

In CRdata we have many different REST calls, in this simple explanation I will use processing nodes and jobs queues as a basic simple example. In order to have a distributed system one has to register all of their processing nodes with the server. The way to do this is to set up a processing node.

To see a list of all of the processing node the REST call is:

http://www.crdata.org/processing_nodes

After you set up a processing node you can view the individual processing node with this REST call.

http://www.crdata.org/processing_nodes/2

The jobs in the CRdata system sit inside a queue. Every time a processing node is sitting idle with nothing to do it asks the server for more work. The server then hands a piece of work to the processing node.

To see a list of all of the job queues you would make this REST call.

http://www.crdata.org/jobs_queues

To view an individual queue you could send this REST call.

http://www.crdata.org/jobs_queues/1

As you can see from the above examples, everything has an unique global ID. The ID just happens to be a URI, thats the way REST works.

Use Standard Methods

The web and REST are based on four standard methods also called the standard web interface. These *verbs* as they call them are GET, PUT, POST, and DELETE. When you enter an ID, or URI/URL into a web browser and hit return, you are actually performing the verb GET.

In REST the verbs are exactly the same, and therefore REST is identical to the web verbs.

Communicate Statelessly

This means that the server keeps no state between requests coming from the client. This is the most brilliant architectural point of the web and allows it to scale up. It also allows scalability across multiple web servers. If you are Google serving up billions of email messages a day from millions of web browsers / clients. It certainly makes it easy that every request coming into the gmail servers do not always have to land on the same server. If a set of servers are busy between requests, then they can be off loaded to other servers. REST works in the same fashion. When you do a GET or PUT to a server, many different servers can handles these individual requests across a set of requests. This is because each request is stateless.

CRdata Implementation

The following sections outline implementation details of CRdata and the motivation behind them.

RubyGems

RubyGems provides a centralized, standard package manager for the Ruby programming language software libraries similar to apt-get in Linux and CPAN in Perl in a self-contained format called a gem. We will outline and describe here the core Gems that are used in CRdata.

Cucumber

Cucumber is a testing framework that is used by CRdata.

RightAws

This is the interface to Amazon EC2 and S3.

Amazon Web Services

From a software architecture and design perspective, Amazon Web Services (**AWS**) is the most important piece of technology that underlies CRdata and for that reason we will expand on its significance and the underlying principles behind it. Today, everyone knows that cloud computing is the major buzz behind innovation and scaling of computational services. Some may argue that its overblown, but we at CRdata believe that its at the heart of the future of computation and that not having it would put you at a serious disadvantage.

We have chosen Amazon as our cloud computing vendor because they are the leader in the field and through out our experience with their service over the past year have been extremely impressed with the reliability, cost, ease of use, and scaling potential.

Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (**EC2**) is a web service that allows you to launch and manage instances. An instance is a Linux virtual machine that acts just like a normal Linux computer sitting on your desk but instead this computer is sitting in the cloud on Amazon servers at certain locations spread throughout the world. When you launch the instance you can decide where on the planet you want this virtual machine to exist and from that point forward any time you log into that computer it is physically located in the place on the planet where you designated the launch.

When you launch an instance you get a choice of the size of instance you want. The size of the instance is a function of both the processor speed and the memory footprint. The process is elastic in the respect that you can launch as many instances as you need at a particular point in time and then later bring all or some of those instances back down.

Amazon Machine Images

Amazon Machine Images (**AMI**) are the actual underlying piece of software that gets run when you launch an Amazon instance. AMI's get created from many different sources but usually an AMI starts out as an operating system and then gets extended from there. As an example, let us say we choose to launch for starters the Ubuntu operating system. When we launch the instance, we are literally choosing the Ubuntu AMI from the list of publicly available AMI's on the Amazon web site. Once our instance is up and running we are starting out with a clean base instance. From there, one can add software to the base instance. In the case of CRdata we add many different pieces of software including Ruby, Ruby on Rails, web servers etc. At that point, after we have added all of the software that makes up CRdata we are ready to create a new AMI that encapsulates the base core that we started out with plus all of the additional software that we added since the time of instance launch.

There are a set of commands that the user goes through in order to create a new AMI, and once it is created it is a software snap shot of everything you need to be up and running with a particular set of software on top of a core operating system. At CRdata we will have available to our users a set of AMI's that enable you to launch both the CRdata platform / server and also a different AMI for the processing nodes.

Amazon Elastic IP Addresses

Amazon Elastic IP Addresses are static IP addresses that are associated with your Amazon account and not a particular instance that you just launched. When an instance gets launched Amazon assigns a dynamic IP address to that running instance. Unlike traditional static IP addresses, however, Elastic IP addresses allow you to mask instance failures by programmatically remapping your public IP addresses to any instance in your account. In order to maintain a static IP address across different launches of your instance over a period of time you can assign the static IP address to the running instance and maintain integrity with your ISP that hosts your domain name. So whoever serves up your domain name you configure your static IP once that Amazon assigns to you and then you are good even though you may bring up and down

Amazon Management Console

The amazon management console provides a graphical user interface that runs in your web browser for managing Amazon instances. It should be noted that this is not the only way to manage instances as there is a corresponding command line reference to each and every command that is available in the console. Common things to do in the console include starting instances, stopping instances and getting a status of a currently running instance.

When you log into the console you will see some top level functionality which helps guide you from the initial point. The top level functionality is called the EC2 Dashboard. The MyResources section shows you how many instances you have running. If you click on that it will show you the individual instances that are running. If you select one of the instances it will show you the Amazon Machine Image (AMI) Id, the status of the running instance, the time that the instance was launched, the zone or place on the planet where the instance is running, and both the public and private DNS address. You can also get information on what Elastic IP's you have assigned, EBS volumes, and your security key pairs just to name a few things.

Amazon Language Bindings and Gems

Amazon provides different programming language bindings to talk to the Amazon web services. The most popular languages are supported including Java, Ruby, PHP, and Python. In CRdata we use the Ruby binding and talk to S3 by a supported Ruby Gem called RightAws.

In conclusion, CRdata runs out of the box on AWS. We chose AWS because of its ease of use, low cost, and simple configuration. When we were first deciding on where to run CRdata we looked around for different hosting providers. We even considered hosting the application on our own computers at Caltech. But in the end, the cost benefit analysis led us to Amazon as the simplest, cleanest solution to running multiple nodes in a seamless fashion.

Data Storage

CRdata uses several different mechanisms for data storage in the system. The most standard type of storage is the relational database which is tied to Ruby On Rails Active Record. CRdata uses the relational database platform PostgreSQL out of the box and we recommend you do as well if you are installing CRdata on your own.

Database

All of the models in **MVC** are stored in the database. Reviewing the models mentioned above you will note that there is quite a bit of information that CRdata keeps track of as the application runs. Probably the most important thing to keep in mind is understanding where data gets stored that the user uploads to the system. This is the data that a CRdata programmer should understand where it lives and how it gets accessed.

R scripts that a user uploads are stored in the relational database and data files that the user uploads are stored in S3. The rational behind this design decision makes quite a bit of sense from a data management and data size point of view. For the most part, R scripts are going to be relatively small compared to the size of a user's data sets. The R script is simply the program that gets run and contains Ascii characters that is easy to store quickly and succinctly in a database. Data files can be very large and contain multiple different types of formats, characters, etc. and so storing it in Amazon's S3 storage system makes the most sense. Keep in mind that the file size limit for S3 files is five (5) Gigabytes.

Amazon Simple Storage Service (S3)

Amazon S3 is a robust, scalable, simple way to store data on the internet. Its the same exact service that Amazon uses to store their own data of books, electronic devices, etc. When you visit an Amazon web site you are using S3. The API is simple, its allows you to read, write and delete objects similar to the way Active Record works and other object database systems.

The number of objects you can store on S3 is unlimited and each object can be referenced through a typical URL style identifier. Its rather convenient to store web pages in S3 because any time you reference the web page in your browser you are literally viewing the data object stored on S3.

Authentication mechanisms are in place for each stored object, and by default the only one who can read an object is the person who stored it. Just like files in a file system, there are ways to assign a group or list of users who have write or update privileges. If you have a set of web pages that are stored in S3, then usually you would give permission to everyone to read the web page, but the ability to update or delete the web page.

Amazon provides a highly durable storage infrastructure where objects are redundantly stored on multiple devices across the world. There are also numerous mechanisms to store your data and then retrieve it later. Most popular programming languages have bindings that allow one to store data through simple to use APIs. There is also a very nice "addon" in Firefox that allows you to upload data to S3 and then download data later from S3 including a synchronization feature where you can sync up data from both sides.

Each object is stored in what is called a bucket. The default CRdata bucket is called *crdataapp.* It is the root location or bucket name where all CRdata is stored. The hierarchy for the data is broken down into three directories which live underneath *crdataapp.*

- results
- logs
- data

Both the results and logs directory are organized according to job numbers. The job numbers are automatically assigned starting from number zero. Each time a new job gets submitted or executed CRdata looks at the ID of the next job number which is persisted in the database and creates two directories in S3. One is a log file which gives the history of the job and whether there were any errors, and the other is the results file which stores all of the associated data including pictures and HTML that is associated with a successfully completed job.

An example of this follows for the results of job number 0.

http://crdataapp.s3.amazonaws.com/results/job_0000000000/index.html

An example of this follows for the log file of job number 0.

http://crdataapp.s3.amazonaws.com/logs/job_0000000000/job.log

The data files that a user uploads is also stored in the bucket *crdataapp* except the URL storage uses a slightly different format than results and logs.

http://crdataapp.s3.amazonaws.com/data/d71da9d0-1990-012d-c917-001ec95a2d04/integers.dat

In the case of data a unique ID is assigned that is stored in the database which identifies a particular data file. The reason that this data does not use the convention of job numbers is because data files can be used across many different R scripts and job submissions and so it uses its own methodology for storage in S3.

Amazon Elastic Block Store

Amazon Elastic Block Store (**EBS**) offers persistent storage for Amazon EC2 instances. The nice feature about EBS is that data persists across the instance life cycle. So for example, you can bring up and instance and start running your web server. As that web server runs it does all sorts of things including updating user records, creating new accounts, accepting new R scripts etc. If at a later point in time you bring that instance down or for some reason it crashes your data is persisted.

The actual physical database files that store all of your information are persisted as you continue to launch and bring down Linux processes. The way EBS works is that you attach the volume to a particular instance when it comes up. Then it is available to that instance as long as the instance runs. You can also detach instance A and re-attach instance B if you so desire. It allows you to basically mount and unmount file systems just like you would on a regular Linux operating system.

Amazon EBS volumes offer improved durability over local Amazon EC2 instance stores, as Amazon EBS volumes are automatically replicated on the backend. A local instance store is the disk space that you get when you launch an instance by providing a mount point for other information as well. For those wanting even more durability, Amazon EBS provides the ability to snapshot your volume to S3 at designated points in time which provides another layer of backup.

Processing Node

CRdata seamlessly integrates multiple processing nodes. A processing node is a computer process that gets launched any where in the world. After registering your processing node with CRdata it is available to do work. In this particular application, the work that gets done is to run an R script. That R script can have associated with it a data file on which to process information based on the R algorithm that one submits.

The processing Node usually runs on a different computer or processor than the server, but it doesn't have to.

The command for bringing up a processing node is

```
ruby processing_node.rb webserver-machine-name.mydomain.com:3000
```

Note that the name of the computer the webserver is running on is called webserver-machine-name.mydomain.com and 3000 is the default port upon which the server is running. You can run as many processing nodes as you like, and each one of them gets launched with the exact same command. Each one of the processing nodes has to be pre-configured on the the server by way of the REST call. In order to do this configuration you must do the following.

This is a completely different concept and although its written in the Ruby Programming Language does not have anything to do with Rails and or Web programming. This is a client side application that talks back to a server that understands REST.

The CRdata server has no idea who it is talking to. This processing node could appear to the server just like another web browser, all the server knows is that it is getting requests sent to it and it understands them because the client is talking REST. That is the beauty of Ruby on Rails. It gives you a free distributed computing platform by baking into the product a REST communications protocol. If we were writing CRdata in another programming language we would have to design our own REST protocol and implement software that makes all of the REST communication happen.

The processing node design is rather beautiful from the point of view that the scaling architecture of the CRdata server in concert with a set of processing nodes is very clean. Imagine a system where you have many processing nodes and many jobs being processed on the CRdata server. The processing node simply sits there and asks the server, "Do you have any work for me to do ?" It just keeps doing that sitting in an infinite loop. Eventually, the CRdata server when it gets a job submitted to it passes off some work for the processing node to do.

So, the scalability of the system is simple. The CRdata server doesn't care who it passes its work to, as long as the processing node meets some simple criteria. First of all, it has to be a registered processing node. The processing node has to have been registered on the server. The processing node can be a node that either processes big jobs or small jobs and thats it.

The server passes to the processing node a job ID.

The processing node is an extremely simple architecture and design compared to the CRdata server.

There are four (4) files in this system.

processing node, job, instrument developer, and global

processing node

This is where everything gets started, in order to launch a processing node you simply type this command

ruby processing_node.rb webserver-machine-name.mydomain.com

or if you are running on port 3000 when your server started

ruby processing_node.rb websever-machine-name.mydomain.com:3000

The processing node kicks everything off and is responsible for the main programming loop. This loop is an infinite loop that sits there forever just asking CRdata if it has any work for it to do. There is a sleep call inside the loop that gets fired off every **n** seconds. In Ruby **n** is in seconds.

The default method sets the sleep time at 1 second. If for some reason you want to change the REST call to increase or decrease the amount of time that the processing node waits before making another request for work simply edit this line with the number of seconds you are interested in.

For example

sleep(1) sleeps for 1 second
sleep(5) sleeps for 5 seconds
sleep(0.1) sleep for 0.1 seconds

Two other things happen inside the processing node including fetching the next job and making a REST call back to the CRdata server telling it that the job has completed and whether it completed in a successful state or a failed state.

Job

When there is actually work (job) sitting in a queue on the CRdata server then when the processing node asks for work a job gets initialized from the XML response that gets returned from the CRdata server. Because the call is a REST call, the data always gets returned in the form of XML.

The XML response is parsed by a Ruby GEM called Hpricot, see the reference section on GEMS for further details.
The XML response contains the r script that the processing node will execute. The CRdata server pulls the r script out of the database and then passes this script back to the processing node, from now on called [**pn**].

The **pn** then checks to see if there are any parameters associated with the R script. One of the params could possibly be a data file that gets processed along with the R script. If there is a data file, then the job will grab the data out of Amazon's S3 system. All data files in CRdata get stored in S3. Any other parameters besides data files are also parsed from the XML response and dealt with accordingly. These other parameters are input parameters that the user defines when setting up an R script that will be run.

Instrument Developer

The instrument developer script core technology is used to tell CRdata what objects to output when an R script is executed. The author of the script must tag the R code with output identifiers and any associated formatting requirements. The Instrument Developer will translate these tags using R2HTML.

This basis of this is regular expressions along with a tag library. For details on the different tags in the tag library see the "*CRdata Instrumentation Insider*" document. This document will explain the details of each individual tag and how to use and define it.

The instrument developers main job is to parse the tags with regular expressions and then store all of the tags into an array. This array is then processed and used by the R library **R2HTML**. For further details on R2HTML see the reference section.

Global

The global class contains many of the global parameters that are needed in the **pn** including the state of the jobs along with names for the temporary directories. The Global class also includes miscellaneous and sundry simple functions that are used across the client.

Queues and Processing Nodes

Any user can create a new Queue. When they create the queue they can choose the Visibility which allows them to set it to private or share. If that particular user happens to be a 'site admin' then that user will be able to create public queues as well. When you create a queue to be shared then you get to select which groups have access to those queues as well. You will only see a list of groups that you are a member of.

After you have created a queue then you can go ahead and add in a new processing node. Upon adding a new processing node you will be required to choose the queue you want to use. You can select to use the queue you may have just created and/or add a new queue that you want to use.

Directory Structure

The top level directories in CRdata are a mirror image of what you would find in any Rails application with two exceptions. There are two extra directories called **features** and **spec**.

app

This is the main code directory and the location of the model, view, controller, and helpers.

config

The database.yml resides here and is the place where you configure your postgresql username and password. The environment.rb file contains gem configuration information along with the schema format for active record. Finally, routes.rb allows for customization of the REST calls that are mission critical to the way CRdata works.

db

There are two main items in this directory. From an initialization and testing perspective there is a file called seeds.rb which allows you to automatically, if you choose to set up your system to come out of the box with some predefined parameters. You do not have to use these seeds, but can bring up a system "blank" and then manually add in the appropriate data according to your system specifications.

doc
The doc directory contains any relevant documentation that may apply to your system.

features
This directory is used solely for configuration of the Cucumber gem.

lib
There is a tasks directory in here that controls cucumber.

public

This directory contains the top level structure of the main web site including flash, images, javascript, and style sheet*s.*

script

This is the standard script directory in all rails applications that allows you to run things like an out of the box web server etc...

test
This directory contains all of the unit tests.

vendor

This directory contains all of the gems that CRdata uses including {put a list here}. Also in this directory is the reference rails implementation that we are using along with the plugins {put a list here}.

Technical Details

Home Page

The home page is where everything starts and when you go to CRdata.org this is the page you land on. It is controlled by the StaticController which also controls the about page and the tour page.

```
class StaticController < ApplicationController
  def index
  end

  def tour
  end

  def about
  end

  def help
  end

  def guest
  end

end
```

The StaticController code is very simple as shown above and its in the corresponding views located in
app\views\static where the static web pages get processed.

Jobs

When you click on the "Run Analyses" tab it takes you to the JobsController index.

```
  def index
    respond_to do |format|
      format.html
      format.js   { render :partial => 'list', :locals => { :jobs => Job.get_jobs(current_user, params) } }
      format.xml  { render :xml => Job.get_jobs(current_user, params.merge({:show => 'all'})) }
    end
  end
```

As you can see from the code above if the request comes from a browser you get the web page and if it comes from a REST call then the XML gets returned. You can simulate this in your browser by typing jobs.xml instead of just the normal jobs extension.

From the 'Run analyses' tab if you click on Add Job it takes you to the New Job page which has a URL of /jobs/new. In the jobs_controller an example of this code is here.

```
  def new
    @job = Job.new

    respond_to do |format|
    format.html # new.html.erb
    format.xml  { render :xml => @job }
  end
end
```

State Machine

The jobs go through a set of states from the beginning when the job gets submitted until the end when the user is able to view the results... The states include {New, Submitted, Running, Done}.

Because of these two lines of code in the Job the state initially gets set to New and the UUID gets set when you add a job.

```
before_validation_on_create :set_to_new
after_create :set_uuid
```

Here is the code that sets the state to new.

```
# Set it to new state if not given already
def set_to_new
  self.status ||= 'new'
end
```

Here is the code that sets the UUID.

```
# Set the job unique identifier
def set_uuid
  self.uuid = "#{id}-#{Digest::SHA1.hexdigest(UUID.generate + Time.now.to_s)}"
  self.save
end
```

Here is how the Job gets queued up when it first gets submitted.

```
# Queue it
def submit(queue = nil)
  self.transaction do
    self.lock!
    # We default to the default queue!!!
    self.jobs_queue = queue || JobsQueue.default_queue
    self.status = 'submitted'
    self.submitted_at = Time.now
    save
  end
end
```

We will go over the round trip job completion code base of what happens to a job from the time it gets submitted to completion when you see the results.

The models or database tables involved with the process of submitting a job all the way to completion which is the viewing of actual results are the following.

JobsQueue for this example we will call the queue "Default Queue"

JobsQueuesController has the following method names that are the standard ones including

index, show, new, edit, create, update, destroy, destroy_all

But the one that is most important includes **run_next_job**. This gets called by the ProcessingNode as a REST call.

```
def run_next_job
  @processing_node = ProcessingNode.find_by_ip_address_and_status(request.remote_ip, 'activated')
  respond_to do |format|
    if @processing_node and @job = @processing_node.jobs_queue.run_next_job(@processing_node)
      flash[:notice] = 'Job was successfully started.'
      format.html { redirect_to(@job) }
      format.xml  { render :xml => { :job => @job, :r_script => @job.r_script, 'params' => @job.job_parameters } }
    else
      format.html { redirect_to :action => 'index', :status => 404 }
      format.xml  { render :xml => 'No job waiting', :status => 404 }
    end
  end
end
```

Inside this method is another call that calls into the model JobsQueue

```
# Get the next scheduled job and return it. nil if none available or error
def run_next_job(proc_node)
  job = nil
  get_next_job = true
  self.transaction do
    while get_next_job
      begin
        job = jobs.first
        job.run(proc_node) if job
        get_next_job = false
      rescue => ex
        # Just name sure we go on
        logger.info "Exception getting next job from queue: #{ex}"
      end
    end
  end
  job
end
```

And then finally the call into the model job is where the job actually gets unqueued and started.

```
# Start running - unqueue and execute
def run(proc_node)
  self.transaction do
    self.lock!
    unqueue('running')
    self.processing_node = proc_node
    self.started_at = Time.now
    save! # this is one instance we really want an exception
  end
end
```

And this is where the model job is finally completed.

```
# Finish run
def done(success)
  self.transaction do
    self.lock!
    self.status = 'done'
    self.successful = success
    self.completed_at = Time.now
    save
  end
end
```

Cloned jobs implement the following code

```
# Clone the job (without saving it)
def cloned_job
  self.transaction do
    j = Job.new( :description =>  self.description, :status => 'new', :r_script => self.r_script)
    j.parameters << self.parameters
    j.job_parameters << self.job_parameters

    j
  end
end
```

When a job gets cancelled this is what happens

```
# Cancel run
def cancel
```

```
    self.transaction do
      self.lock!
      if jobs_queue
        unqueue('cancelled')
      else
        self.status = 'cancelled'
      end

      self.successful = false
      self.completed_at = Time.now
      save
    end
  end
```

Each time a state transition happens there is a check to make sure the transition is valid

```
  # Make sure the state we are transitioing to is valid!
  def check_state_transition
    # List of statuses we are allowed to set. Anything else is an error
    $ALLOWED_STATUSES ||= { nil => ['new'], 'new' => ['submitted', 'cancelled'], 'submitted' => ['running',
'cancelled', 'new'],
                            'running' => ['done', 'cancelled', 'new'], 'done' => ['new'], 'cancelled' => ['new'] }
    errors.add(:status, 'Invalid status transition!') if status_changed? &&
!$ALLOWED_STATUSES[status_change[0]].include?(status_change[1])
    # We cannot change the submitted_at if it's already set
    errors.add(:submitted_at, 'Cannot resubmit the same job') if submitted_at_changed? &&
!submitted_at_change[0].nil?

    # We cannot change the completed_at if it's already set
    errors.add(:completed_at, 'Cannot complete the same job') if completed_at_changed? &&
!completed_at_change[0].nil?
  end
```

CRdata How To's

This section describes **How To's** that better explain the major parts of CRData that need further details and are fairly well encapsulated as individual topics. As you read through this document, if there is another How To document that you feel is important and would like to see it let us know and we will attempt to clarify the topic for you in a How To document.

How to Install CRdata on Amazon EC2

If you want to know the gory details of how we built the CRdata public AMI from scratch then read past the section on installing CRdata from a public

AMI. However, the simplest way to install CRdata is to launch your own Amazon instance.

How to Setup an Amazon Security Group

In the Amazon AWS Management Console setup a new Security Group with these settings. For example purposes we will call this security group **demo**. So whenever you see **demo** in this documentation you will know it is referring to the security group.

Connection Method	Protocol	From Port	To Port	Source (IP or Group)
SSH	tcp	22	22	0.0.0.0/0
-	tcp	5432	5432	1.2.3.4/0
HTTP	tcp	80	80	0.0.0.0/0

Conventions used in these Amazon examples

In all of these examples the IP address is 1.2.3.4, so whenever you see 1.2.3.4 you will know you can insert your own specific IP address. Associated with the IP address 1.2.3.4 is the Amazon DNS name

ec2-1-2-3-4.compute-1.amazonaws.com

When ever you see this IP address you can insert your own IP address that you can obtain from Amazon once you launch your AMI. The IP address is located in Amazon's Management Console.

Keypair name is used by the command ec2-run-instances and in these examples is called keypair_name and is created with the following command.

```
ec2-add-keypair keypair_name
```

A new 2048 bit RSA key pair is created with the specified name. The public key is stored by Amazon EC2 and the private key is displayed on the console. The private key is returned as an unencrypted PEM encoded private key which in these examples will be called keypair_name.pem

This section is from the Amazon documentation and is here to help familiarize you with how Amazon security works. This is the simple explanation so that you understand the security files we are are discussing here. For further details please consult the Amazon EC2 documentation.

Amazon gives you the ability to generate a X.509 certificates consisting of a certificate file and a private key file:

- **X.509 Certificate—**The certificate holds the public key and related metadata. You include it in each service request, so it's not a secret.
- **Private Key—**Each certificate has a private key associated with it. Use the private key to calculate the digital signature to include in the request. Your private key is a secret, and only you should have it. AWS doesn't keep a copy of it.

When you create a request, you create a digital signature with your private key and include it in the request, along with your certificate. When Amazon gets the request, they use the public key in the certificate to decrypt the signature and confirm that you're the request sender. Amazon also verifies that the certificate you provide matches the one they have on file.

If you have Amazon generate your X.509 certificates, they give you a PEM-encoded certificate file, and a PEM-encoded private key (unencrypted, which means you don't get a password for it).

If you provide your own keys, you must upload only your certificate to AWS (you keep the private key). The certificate must be in PEM format. AWS accepts any syntactically and cryptographically valid, unexpired X.509 certificates. They don't need to be from a formal Certificate Authority (CA).

How to Build and Bundle an Amazon AMI from scratch.

We will assume that your Amazon instance is launched and that you have ssh'd into it.

On your local machine, not on the Amazon instance.

```
scp -i keypair_name.pem X.509cert.pem private_key.pem
 root@ec2-1-2-3-4.compute-1.amazonaws.com:/mnt
scp -i keypair_name.pem cert-[   ].pem  pk-[   ].pem
 root@ec2-1-2-3-4.compute-1.amazonaws.com:/mnt
```

Copying the files to /mnt is important because when the AMI gets built these key files don't get copied over to the AMI. Files that are located in /tmp however do get copied over, so do not copy the files to /tmp.

On your Amazon instance run these commands.

ec2-bundle-vol
Creates a bundled AMI by compressing, encrypting and signing a snapshot of the local machine's root file system.

ec2-upload-bundle
Uploads a bundled AMI to Amazon S3 storage

ec2-register
Registers an AMI with Amazon EC2. Images must be registered before they can be launched.

```
ec2-bundle-vol -k pk-[   ].pem  -c cert-[   ].pem -u [amazon account number w/o dashes]
```

It will come back with a message that says:

Please specify a value for arch [i386]:

At this point simply hit return and then the process will begin...

```
ec2-upload-bundle -b bucket -m /tmp/image.manifest.xml -a <access key> -s <secret key>
```

ec2-upload-bundle -b **your-bucket-name** -m /tmp/image.manifest.xml -a <access key> -s <secret key>

OR wherever the image.manifest.xml file is located, this is a real file that you must reference so where ever you put it thats what you want to reference above. Also note that if the bucket does NOT exist Amazon will create the bucket for you.

Back on your local machine go ahead and register the AMI image.

```
ec2-register your-bucket-name/image.manifest.xml
```

or if you don't have the Amazon tools installed on your local box then go to the **AMI** tab in the Amazon Management Console and register the new AMI by putting in the box that comes up.

your-bucket-name/image.manifest.xml

Note after this AMI is built the data located in /tmp is still there, so if you put ZIP or TAR files and unzip or untar them in the /tmp directory they will still be there. In other words, once again, do NOT store your security files like certifcates, private keys etc... in /tmp, instead copy them to /mnt.

That is why up above the keys get copied to /mnt, because that data is gone when you fire up the new AMI.

For your reference here are a couple of other relevant things to do.

When you are no longer interested in keeping around a private AMI simply go to the AWS Management Console and go to the AMI tabs and **deregister** the particular AMI's you no longer want.

After doing that go into the S3 Firefox Organizer and delete the associated S3 buckets with the AMI's you just deregistered.

The details of building a CRdata public AMI

The following sections outline in vivid detail all of the steps needed to build a CRdata AMI.

Installing Ruby on Rails on your Amazon Instance

These instructions will take you from an off the shelf bare Right Scale CentOs to a system running with Ruby on Rails. CentOS stands for the Community Enterprise Operating System and is the Open Source version of Red Hat Enterprise Linux.

These are instructions to go from the off the shelf RightScale CentOs 5.4 AMI to a completely working system with

Ruby 1.8.7
Rails 2.3.5
Gem 1.3.6

Here is the web site for the RightScale Amazon Machine Image with this AMI id ami-f8b35e91.

To launch the CentOs

```
ec2-run-instances ami-f8b35e91 -k keypair_name --availability-zone us-east-1b
```

Once the instance is launched you can SSH into the image with this command.

```
ssh -i keypair_name.pem root@ec2-1-2-3-4.compute-1.amazonaws.com
```

Now that you are ssh'd into your machine you can update your operating system to the latest and greatest code base by issuing this command

```
yum update
```

If you prefer to use emacs as your editor instead of vi you can issue this command.

```
yum install emacs
```

This will show you the default ruby tools that are installed on your machine.

```
cat /var/log/yum.log | grep ruby
rpm -qa | grep ruby
```

Now you can go ahead and erase all of the old ruby code that is installed on the machine in preparation for installation of the new Ruby tools.

```
yum erase ruby ruby-libs ruby-irb ruby-rdoc ruby-ri ruby-mode ruby-devel
ruby-docs ruby-tcltk rrdtool-ruby
```

Make sure its all gone

```
cat /var/log/yum.log | grep ruby
rpm -qa | grep ruby
```

The **openssl-devel** C library is required so that when you build Ruby from scratch you are able to link in the Open SSL library. These tools are required by the Amazon EC2 command line tools that are installed on the default RightScale image.

```
yum install openssl-devel
```

Now grab Ruby.

```
cd /tmp; wget ftp://ftp.ruby-lang.org/pub/ruby/1.8/ruby-1.8.7-p72.tar.gz
```

Now you have the zipped source code. This will unzip it...
```
tar -xzvf ruby-1.8.7-p72.tar.gz
```

Go to the directory that it unzipped to...
```
cd ruby-1.8.7-p72
```

This command will run a script for the make file that is used to compile and install Ruby.

```
./configure --with-openssl
```

Now we run the make.

```
make
make install
```

Ruby is installed. Let's get Ruby Gems...

```
wget http://rubyforge.org/frs/download.php/56227/rubygems-1.3.3.tgz
```

```
tar -xzvf rubygems-1.3.3.tgz
```

Now go into the directory and run this command.

```
ruby setup.rb
```

The following command takes you up to gem 3.6.

```
gem update --system
```

Rails is part of the CRdata product so it doesn't have to get installed, but if you want it installed on your machine then you can issue the following command.

```
gem install -v=2.3.5 rails
```

At this point the following products are installed.

Ruby 1.8.7
Rails 2.3.5
Gem 1.3.6

To test to make sure the versions are properly installed you can issue the following command.

```
ruby -v
rails -v
gem -v
```

Now you can make an Amazon AMI with your new code base if you like or keep going with the instructions. The advantage of making an intermediate Amazon AMI is that you have a snapshot of just the Bare CentOS along with Ruby et al. This way if you make a mistake going forward from here. You don't have to start all over again, but can start from this snapshot point. It also will give you some good practice for building an Amazon AMI.

Installing the PostgreSQL Database on your Amazon instance

Install the latest Postgres (currently is 8.4)
wget http://yum.pgsqlrpms.org/reporpms/8.4/pgdg-centos-8.4-2.noarch.rpm
rpm -Uhv pgdg-centos-8.4-2.noarch.rpm
yum install postgresql postgresql-devel postgresql-server

Configure Postgres

sudo /etc/init.d/postgresql initdb [this command takes about one minute so be patient]

Edit /var/lib/pgsql/data/pg_hba.conf

the last 3 lines have a method name called '**ident**' change that word to '**trust**'

sudo /etc/init.d/postgresql start

Set a password for the postgres user
psql -U postgres
alter user postgres with password 'pg_root';
\q

Edit /var/lib/pgsql/data/pg_hba.conf

the last 3 lines have a method named called '**trust'** change that word to '**md5**'

Restart the server
sudo /etc/init.d/postgresql restart

and make it autostart on load
chkconfig postgresql on

Setup the crdata db + user

psql -U postgres # Enter the password when prompted which is 'pg_root'
create user crdata with password 'crdata123';
create database crdata owner crdata encoding 'UTF-8' template template0;
\c crdata
create language plpgsql;

```
\q
```

How to talk to your Amazon PosgreSQL database through pgAdmin III

Make sure you set up a security group from the above instructions so that port 5432 is open.

tcp port number 5432 should be open for connections from pgAdmin III

This should only be opened up for demo purposes.
On a live running server instance open to the public this port should NOT be open for security reasons.

Set up an elastic IP address and associate with your running Amazon instance.

To configure remote access

Edit your PostgreSQL

By editing these files you will be able to talk to your database through the pgAdmin III User Interface.

```
edit /var/lib/pgsql/data/postgresql.conf
```

search for 'listen_addresses' and add the next line into this file

```
listen_addresses = ‘*’
```

uncomment the line

```
port = 5432
```

`edit /var/lib/pgsql/data/pg_hba.conf` and

In the last line of the file modify the IP address to point to your Windows machine running pgAdmin III.
Add in the next line --- to the last line of the file pg_hba.conf

```
host all all 1.2.3.4/32 trust
```

where 1.2.3.4/32 is the IP address of your windows machine running pgAdmin III

Restart your database on your Amazon instance.

```
service postgresql restart
```

Installing the CRdata software on your Amazon instance

```
cd /tmp
```

```
wget http://crdatacode.s3.amazonaws.com/crdata20100413-nginx.tar.gz
tar xzf crdata20100413-nginx.tar.gz
```

cd into the directory that was created called crdata and move the crdata directory to root /

```
mv crdata /
```

Change directory to root / and modify the permissions on the /crdata directory

```
cd /
chown root crdata
chgrp root crdata
```

Gem List

CRdata relies on a lot of Ruby gems, here is a list of what Gems are installed at different times

This gem gets installed when you initially install the Ruby Gems system.
rubygems-update (1.3.6)

Part of the Initial Rails Install

actionmailer (2.3.5)
actionpack (2.3.5)
activerecord (2.3.5)
activeresource (2.3.5)
activesupport (2.3.5)
rack (1.0.1)
rails (2.3.5)
rake (0.8.7)

Install the PosgreSQL gem
postgres (0.7.9.2008.01.28)

```
gem install postgres
```

Check to make sure it got installed and is in the list.

```
gem list
```

After grabbing the CRdata software above, go to the CRdata software directory and run this command to pull down the other Ruby gems automatically.

```
rake gems:install
```

```
aws-s3-0.6.2            [IGNORE the error message on this gem]
builder-2.1.2
mime-types-1.16
right_aws-1.10.0
right_http_connection-1.2.4
xml-simple-1.0.12
```

Here is a complete list of gems that should be installed on your machine.

```
actionmailer (2.3.5)
actionpack (2.3.5)
activerecord (2.3.5)
activeresource (2.3.5)
activesupport (2.3.5)
aws-s3 (0.6.2)
builder (2.1.2)
mime-types (1.16)
postgres (0.7.9.2008.01.28)
rack (1.0.1)
rails (2.3.5)
rake (0.8.7)
right_aws (1.10.0)
right_http_connection (1.2.4)
rubygems-update (1.3.6)
xml-simple (1.0.12)
```

Configuring CRdata

In the main CRdata directory run the following two commands

```
rake db:migrate
rake db:seed
```

See the section above on how to set up your Amazon S3 bucket. This will enable you to see and store your results. If when you run a job and you get an "ACCESS DENIED" when trying to view your results or log file then your bucket is not set up.

Add to your system environment variables the following 2 Amazon keys for your security credentials panel...

```
export CRDATA_ACCESS_KEY='XXYYZZ123'
export CRDATA_SECRET_KEY='AABBCC456'
```

Before bringing up the server on the line below make sure your processing node is running

```
ruby script/server -p 80
```

If things are communicating properly you should see the message coming back from the processing node
in the console of the Ruby Server....

Create a user and set the approved field in the DB and make the user an admin.
Create a group from the UI and call it mydemogroup. Then go to the DB and make it the Default group.

Create a Default Queue and make it public.
Create a Processing Node.

Add your R script and make it public.
Add your job and then submit it.

How to Install CRdata Advanced Features

CRdata comes out of the box with the built in Rails web server called WEBrick. Up until now all of our examples has used WEBrick as our default web server. CRdata works fine with the default web server and full functionality is available through WEBrick. For this reason, and to keep the installation simple and minimal we have decided not to introduce any advanced features until you have become comfortable with the core CRdata technology. By now you should have installed CRdata and brought up the application and run some jobs through the system.

In this Advanced section we will discuss some of the advanced features of installing, configuring and running CRdata. These features include the mongrel web server, mongrel cluster along with the reverse proxy server nginx.

Reverse proxies are used in front of web servers where all connections coming from the Internet addressed to one of the Web servers are routed through the proxy server, which may either deal with the request itself or pass the request wholly or partially to the main web servers. A reverse proxy dispatches in-bound network traffic to a set of servers, presenting a single interface to the caller. Reverse proxies typically can be used for load balancing a cluster of mongrel web servers.

Nginx is also great at serving up static content. So the image, jpeg, gif, and png file types along with css, and index.html are ideal candidates for nginx. The dynamic content is handled by a cluster of mongrel web servers that are configured in the nginx configuration file.

To install the CRdata advanced features simply run the following four (4) commands.

```
gem install mongrel
gem install mongrel_cluster
```

```
rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/epel-
release-5-3.noarch.rpm
yum install nginx
```

then go to the main crdata directory and type

```
mongrel_rails start -p 80
```

this will bring up CRdata in the mongrel web server in a similar fashion to typing ruby script/server -p 80
which brings up CRdata in the simple built in to rails WEBrick server.

In your browser make sure the server is up and running.

Then bring down the server and bring up the cluser.

```
mongrel_rails cluster::start -C /crdata/config/mongrel.yml
```

Now we will configure Nginx

This is the file that starts everything off and the file that is used to run nginx.

```
/etc/init.d/nginx
```

Read this file and use it for configuration...

Inside this file is defined

```
NGINX_CONF_FILE="/etc/nginx/nginx.conf"
```

go to the /crdata/nginx directory and copy these files to their respective locations

denies.list	to	/etc/nginx
nginx.conf	to	/etc/nginx
crdata.conf	to	/etc/nginx/conf.d
crdata.conf.part	to	/etc/nginx/conf.d

Make sure you have set up your Elastic IP on Amazon so you can assign it to the server_name line here.
If you don't know how to set up an Elastic IP read the Amazon documenation.

```
Edit /etc/nginx/conf.d/crdata.conf
```

with the proper name for server_name that you are using from your Amazon console.

Then run the following command

```
/etc/init.d/nginx start
```

Everything should now be up and running under Nginx.

Installing CRdata from a public Amazon AMI

The simplest way to install your own CRdata server is to create your own public AMI, see the detailed instructions in the section above and then launch this public AMI by following these steps.

To launch this public AMI execute this command.

```
ec2-run-instances ami-xxx -k keypair_name -g security_group --availability-zone us-east-1b
```

Once this instance is up and running then go ahead and ssh into the instance. You can get the IP address for this instance from the AWS Management Console.

```
ssh -i keypair_name.pem root@ec2-1-2-3-4.compute-1.amazonaws.com
```

Now that you are logged into your CRdata Amazon instance you can begin to prepare your instance to run the CRdata application.

Edit the name of your Amazon bucket

Edit the name of your Amazon bucket where your CRdata results, logs, and datafiles are stored

In the crdata source code edit a file called

```
/config/initializers/settings.rb
```

Modify the name associated with the variable MAIN_BUCKET to the bucket name you desire by creating a bucket in your Amazon account. You may also choose to leave the name 'crdataapp' as is and simply create the bucket with that name in your Amazon account.

Create a bucket in your Amazon account with a tool called S3Fox Organizer. Its runs in the Firefox Browser and can be downloaded from the Firefox addons website. There are many different tools that enable you to create, edit, and delete Amazon S3 buckets. You can choose any one you are comfortable with, but S3Fox Organizer is the one we use and like and so we recommend that one.

For example purposes I will call the bucket 'mycrdatabucket'. Create an S3 bucket with any name you like. Then underneath that bucket create three directories and call them

data
logs
results

CRdata requires that the directory names under mycrdatabucket have the names {data,log,results}.

Launch a processing node

To simulate a real running CRdata application launch your Processing node on a different computer and point back to the IP address where CRdata is running.

```
ruby processing_node ec2-1-2-3-4.compute-1.amazonaws.com
```

For more details on running a processing node go to that section.

Set your environment variables

Add to your system environment variables the following 2 Amazon keys from your Amazon security credentials panel on your Amazon Account.

CRDATA_ACCESS_KEY
CRDATA_SECRET_KEY

```
export CRDATA_ACCESS_KEY='AABBCC123'
export   CRDATA_SECRET_KEY='XXYYZZ123'
```

Launch the web server

By launching the web server you are bringing up CRdata on port 80 which has to be clearly specified.

```
ruby script/server -p 80
```

If things are communicating properly you should see the messages coming back from the processing node
in the console of the Ruby Server.

Now point your browser at the URL

http://ec2-1-2-3-4.compute-1.amazonaws.com

and you should see CRdata up and running in your browser.

How to Install CRdata on Windows

Installing the CRdata software on Windows

On your windows machine download the software from this URL and store it locally. Then untar it to any location.
http://crdatacode.s3.amazonaws.com/crdata20100413.tar.gz

The setup on Windows is slightly different than the setup on Linux, the main difference being that the database is configured completely in the pgAdmin III. Read through the complete Linux instructions before attempting to install the software on a Windows machine.

We recommend that you go through and install a complete CRdata system on your Amazon account first. Once you are very comfortable with the procedure on Linux, then go ahead and repeat the procedure on Windows with the obvious Linux Windows differences.

Create the database crdata in the pgAdmin III instead of using the command line tools described in the Linux section. Do NOT run any of the command line database commands on Windows. After doing that, everything else should be the same.

Gem List on Windows

CRdata relies on a lot of Ruby gems, here is a list of what Gems should be installed on your Windows machine. The main difference between the gems on Windows and Linux is the gem for PostgreSQL. Note that there is a different gem for Windows and Linux.

ruby-postgres (0.7.1.2006.04.06)

As on the Linux platform, these gems are part of the Initial Rails Install

actionmailer (2.3.5)
actionpack (2.3.5)
activerecord (2.3.5)
activeresource (2.3.5)
activesupport (2.3.5)
rack (1.0.1)
rails (2.3.5)
rake (0.8.7)

And these Gems need to be installed as well for a fully functional system on Windows.

aws-s3 (0.6.2)
builder (2.1.2)
mime-types (1.16)
right_aws (1.10.0)
right_http_connection (1.2.4)
xml-simple (1.0.12)

How to Manually Launch an Amazon R Node Instance

The public AMI id for the R node is found by searching for "r_node" under the Community AMIs when you launch an instance.

In the User Data box that comes up when you launch the R node from the Management console enter the following information

url='http://1.2.3.4' uid = 'anystring' where

1.2.3.4 is the IP address of the running CRdata

Then when asked for a key pair choose the key pair that you have been using and the security group should have the following characteristics.

protocol tcp with port 2222 open.

Connection Method	Protocol	From Port	To Port	Source
-	tcp	2222	2222	0.0.0.0/0

Once the R node comes up then associate an elastic IP address with the running node so that you can then put that IP address into the configuration when adding a Processing Node in CRdata.

How to Understand CRdata's API Using REST

REST is at the heart of the CRdata architecture and implementation. Every single request from all the browser's in the world that hit the CRdata server are talking to it in REST. The beauty of the Processing Node above is that even though the Processing Node is not a browser, it can act like a browser to the CRdata server by sending requests to it via REST. So if you dive into the Processing Node source code you will see that all of the requests to the server are a subset of the following commands.

CRdata REST API Details

Clearly, the Processing Node uses only a subset of the commands above, but by understanding the REST commands in detail there are many different types of things you do with CRdata.

In the next section we will discuss how to use these REST commands to extend CRdata.

How to Extend CRdata to Process Scripts Other Than R

Inherently, CRdata is a distributed computing system where Processing nodes talk back to the CRdata server to get information that allows the

Processing Node to run R programs. All of the infrastructure is in place however to allow one to use this distributed computing platform to implement Processing Nodes that do other things besides process R jobs.

Using the current Processing Node code as a basis for communication and as a model system, one can build a new Processing Node that does completely different type of work. In order to do this, one needs to come up with a set of tasks that you want to do and then define a new REST API that accomplishes the tasks you want to do.

As an example, suppose one wants to have a distributed system that runs financial algorithms that decides when to buy and sell stocks. Each processing node could be charged with running a different type of algorithm depending on the state of the financial markets at some moment in the trading day. The current processing node makes a request to the CRdata server to ask for work and the server sends back to the processing node the R script to run. Instead the CRdata server could send back a portfolio of stocks that the particular equity trading system node may want to run. The only thing that would have to change to do this is the REST API that is defined in the routes.db file. This file, is the central switch that enables communication between the client and the server. On the server side once the API is defined, then when the particular request comes in from the client the server knows about the particular format of that request and handles it appropriately sending back the well defined response.

References

Flanagan, David: The Ruby Programming Language, 1st Edition, 2008 January

Tate, Bruce A.: Rails: Up and Running, 2nd Edition, 2008 October

Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures,*
Doctoral Dissertation, University of California, Irvine, 2000.

Stefan Tilkov's A Brief Introduction to REST

Eric Lecoutre's R2HTML

Gamma, Erich et al: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley

References of Ruby Gems

Hpricot This gem parses XML that gets returned from the CRdata server

Appendix

Various and sundry Nginx commands

See the log file here...
/var/log/nginx/error.log

To see the version of nginx
```
nginx -v
```

To test the configuration file
```
nginx -t
```

```
/etc/init.d/nginx stop
/etc/init.d/nginx start

/etc/init.d/nginx restart
/etc/init.d/nginx reload

/etc/init.d/nginx status
/etc/init.d/nginx configtest
```

Interesting and helpful notes about Ruby

In order to move jobs through their different states we use Ruby Procs which encapsulate local state as a closure.

Interesting and helpful notes about Ruby on Rails

One of the first things you may want to do if you are create a Rails project from scratch is to build the directory hierarchy that we outline above. In order to do that you must first create a Rails project.

A simple command to create a Rails project is

```
rails myfirstproject
```